PPL: A Logic for PTIME Queries on Programs

Anders Miltner
amiltner@seas.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Scott Weinstein
weinstein@cis.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

Val Tannen
val@cis.upenn.edu
Univ. of Pennsylvania
Philadelphia, PA

ABSTRACT
There is a vast number of PTIME queries on graphs, and it is difficult to prove whether a query is or is not a PTIME query. Fagin’s theorem shows that existential second order logic perfectly captures all NPTIME queries on arbitrary structures. Does PTIME have a logic which can do the same?

As this question is very broad, and indeed a negative answer would prove P ≠ NP, the purpose of PPL will not be to prove whether or not PTIME has such a logic. A logic will be provided that perfectly captures all PTIME queries on Java programs. This enables all efficient programs that return true or false to be expressed by a sentence. This is useful for it provides an alternate means to creating efficient programs. This can be useful for determining if an optimization can be applied to a program.

1. BACKGROUND

Some relevant background material is required for an understanding of PPL and the the proof of its expressiveness. This section aims to define and state these relevant terms and theorems.

- A relation \( R \) on \( k \) variables over a set \( X \) could be viewed as a subset, \( R \subseteq X^k \). Equivalently, it could be viewed as a function on \( k \) variables \( R : X^k \to \{ \text{true}, \text{false} \} \). These are equivalent notions, as one could say a tuple \( (x_1, ..., x_k) \in R \) if and only if \( R(x_1, ..., x_k) = \text{true} \).

- A structure (or model) is a set of elements, as well as a set of relations on those elements. For example, the structure with elements \( \{x_1, x_2, x_3\} \) and binary relation \( E = \{(x_1, x_3)\} \) would be the graph with 3 nodes and an edge between the first and third node.

- A class of structures is the set of structures over the same relations. Any class of structures can be encoded into a bit string such that if \( A_1 \) was encoded in the string \( b_1 \), and \( A_2 \) was encoded in the string \( b_2 \), then \( A_1 \neq A_2 \Rightarrow b_1 \neq b_2 \). An example of this would be the class of structures \( G \), the class of graphs. An \( n \)-ary graph can be encoded in an \( n^2 \) long bit string where the \( n \times i + j \)th entry is 1 if \( (i, j) \) is an edge, and false otherwise.

- A query \( Q \) is a function from the set of structures to \{true, false\}.

- For a structure \( A \), and a sentence, \( \phi \), \( A \models \phi \) if \( \phi \) holds true in \( A \).

- A logic \( L \) is a language (i.e. \( L \subseteq \Sigma^* \), where \( \Sigma \) is the alphabet \( L \) is over), along with an interpretation for any element of the language as a function from the set of structures to \{true, false\}.

- \( \phi \) is a sentence in \( L \) if and only if \( \phi \in L \) and for a structure \( A \), \( A \models \phi \) if and only if \( \phi(A) = \text{true} \).

- A logic \( L \) expresses (or captures) a set of queries if for every query \( Q \), there is a sentence \( \phi \in L \) such that for any structure \( A \), \( A \models \phi \) if and only if \( Q(A) \) is true, and furthermore, for every sentence \( \phi \in L \), there is a query \( Q \) such that for any structure \( A \), \( Q(A) \) is true if and only if \( \phi \models A \).

- A fixed point relation \( \mu(\phi)(R) \) is a relation on \( k \) variables, where \( R \) is a k-ary relation, where \( R(a_1, ..., a_k) \) is true only if the sentence \( \phi(a_1, ..., a_k) \) is true, where \( \phi \) is a sentence that uses any of the variables \( x_1, ..., x_k \) as constants, \( R \) as a relation, and the rest of the language as normal. Furthermore, if \( T \) is another relation where \( T(a_1, ..., a_k) \) is true if and only if \( \phi(a_1, ..., a_k) \) is true, then \( R \subseteq T \).

- For a set \( S \), \( 2^S = \{ T \mid T \subseteq S \} \) the set of all subsets of \( A \).

- Given a sentence \( \phi(R) \) with \( k \) free variables, and using the relation \( R \), and a structure \( A \), \( I^\phi_{\lambda}(S) : 2^A^k \to 2^A \) is defined by \( I^\phi_{\lambda}(S) = \{(a_1, ..., a_k) \mid \phi(S)(a_1, ..., a_k) \} \). Intuitively, it is the set of \( k \) tuples that are true of \( \phi \), when \( R \) is taken to be the input \( S \).

- Tarski showed[8] \( \mu(\phi)(R) = \lim_{n \to \infty} (I^\phi_n)^\omega(\emptyset) \). This provides a method to prove statements about fixed point relations by using induction on \((I^\phi_n)^\omega(\emptyset)\).

- Graph connectivity is the query that returns true if, between any two nodes \( A, B \), there is a sequence of points \( A_0, A_1, ..., A_n \) such that \( A = A_0, A_n = B \), and \( (A_i, A_{i+1}) \) is an edge \( \forall i \).

- First Order Logic (FOL) is a logic allowing for \( \exists, \forall \) quantification on elements, and any finite number of \( \land \)s (conjunctions) and \( \lor \)s (disjunctions). An example of a sentence in first order logic expressing that there are no isolated points would be \( \forall x(\exists y(E(x, y))) \).

- First Order Logic with Least Fixed Points (\( FOL + LFP \)) is first order logic allowing for fixed point relations in addition to normal relations. An example of a
sentence in FOL + LFP expressing graph connectivity would be \( \forall x (\forall y (E(x,y) \vee \exists z (E(x,z) \land R(z,y)))(R)) \).

- Immerman-Vardi[5] states that every FOL+LFP query on a structure \( A \), where \( a \) has a binary relation \( \leq \) where \( \leq \) is an antisymmetric (\( a \leq b \) and \( b \leq a \) implies \( a = b \)), transitive (\( a \leq b \), and \( b \leq c \) implies \( a \leq c \), and total (\( a \leq b \) or \( b \leq a \)) relation.

- Given a structure \( A \), \( A^+ \) is defined as \( A \cup \{0, \ldots, |A|\} \).

- FOL+LFP+C on \( A \) is FOL+LFP on \( A^+ \), along with \( \#_a(x) \) which is the function that returns the number of elements in \( A \) that satisfy \( \phi \). FOL + LFP + C captures \( \text{PTIME} \) perfectly on ordered structures, so is equivalent in expressive power to FOL + LFP on them.[3]

- Graph connectivity can not be expressed in FOL, as FOL has a certain limit in scope, where it can not express queries relating to arbitrary length information. In FOL queries, there must be a chosen the length as FOL allows only a finite number of variables in any given sentence. [6] Fixed point logic is an extension to first order logic which allows for these queries that have a transitive nature in them to be expressed. As graph connectivity can be thought of in a transitive way \( A \) is connected to \( B \), if and only if \( E(A,B) \) or there is a \( Y \) such that \( E(Y,Y) \), and \( Y \) is connected to \( B \). FOL + LFP can capture it as LFP relations includes those that can be built in this transitive fashion. One can see the sentence: \( \forall x (\forall y (E(x,y) \vee \exists z (E(x,z) \land R(z,y)))(R)) \) captures graph connectivity, as \( R(x,y) \) is true when \( x \) is connected to \( y \). One can see that this \( R \) is such a relation, as \( R(x,y) \) is true when \( E(x,y) \), or when there is an element \( z \) such that there is an edge between \( x \) and \( z \) (\( E(x,z) \)), and \( z \) is connected to \( y \) (\( R(z,y) \)).

- Existential Second Order Logic (\( E - \text{SOL} \)) is a logic allowing for sentences of the form \( \exists R_0 \ldots R_m(\phi) \) where \( R_i \) are relations, and \( \phi \) is a sentence in FOL using the relations \( R_0 \ldots R_m \), and allowing for the use of relations \( R_0 \ldots R_m \) in addition to the built in relation \( E \). An example of a sentence in ESOL expressing that the size of a graph is even (evenness) would be:
  
  \[
  \exists B \exists S (\\forall x (\\exists y (B(x,y))) \land \\
  \forall x (\\forall y (\\exists z (B(x,y) \land B(x,z) \Rightarrow y = z))) \land \\
  \forall x (\\forall y (\\exists z (B(x,y) \land B(z,y) \Rightarrow z = x))) \land \\
  \forall x (\\forall y (\neg S(x) \land B(x,y) \Rightarrow \neg S(y))) \land \\
  \forall x (\\forall y (S(x) \land B(x,y) \Rightarrow S(y))).
  \]

  Intuitively, this states that there are two relations, \( B \), a relation on 2 variables, and \( S \) a relation on 1 variable, where \( B \) relates every element to exactly one other element, and for every two elements in \( B \), exactly 1 is true in \( S \), and the other is not true in \( S \). This is only true when it is possible to split the elements in 2 groups of equal size, 1 group would be held in \( S \), and 1 would not, and they are equal size as for every element in one group, there is exactly 1 element in the other for it. Being able to split the elements into 2 groups of equal size is the same as saying the number of elements is even.

- \( \text{PTIME} \) queries on graphs are those queries for which there exists a Turing Machine which responds in the same way as the query such that the maximum number of steps that Turing Machine can take must be a polynomial of the size of the graph.

- \( \text{NPTIME} \) queries on graphs are those queries for which there exists a nondeterministic Turing Machine which responds in the same way as the query such that the maximum number of steps that Turing Machine can take must be a polynomial of the size of that graph.

- In figure 1, one can see a diagram of containment of the expressibility of the various logics. These containments, however, are not necessarily strict. For example, were \( P = NP \), then the outer 2 circles would be the same. Further, if the diagram was on the expressibility of various logics on ordered graphs, then the middle 3 circles would be the same, and if \( P = NP \) as well, then all of the 4 outer circles would be the same.

---

**2. INTRODUCTION**

\( \text{PTIME} \) queries are important, as the amount of time they take is always small, relative to queries not in \( \text{PTIME} \). If there are only exponential time algorithms to find a solution, computing that solution is unviable for large graphs. Therefore, demonstrating that a logic perfectly captures \( \text{PTIME} \) would be very useful. If a query had no corresponding sentence in the logic, then it is not viable to attempt to calculate the response of the query on relatively large graphs. In a similar manner, if a query had a corresponding sentence in the logic, then there would be a fast algorithm to compute that query.

First order logic can be expanded capture more queries, with the intent of possibly getting a logic large enough to capture \( \text{PTIME} \) queries, but not anyway else. To look at where the extensions should lie, it makes sense to look at what queries are currently not expressible.

While First Order Logic does not express \( \text{PTIME} \), as queries like evenness cannot be expressed, with additional invariants on the graph, or extensions to the logic, we can gain a deeper understanding of \( \text{PTIME} \), and perhaps get an expressive enough logic. Seeing what sort of sentences are difficult to express in \( \text{PTIME} \), and what sort of structural invariants can make a logic express more, can give a stronger intuition for \( \text{PTIME} \).
Programs are inherently intimately tied to Computer Science. As such, programs are a natural class of structures to attempt to understand. Beyond merely being important to Computer Science, programs have a very set format. This rigid structure allows for greater expressibility in weaker logics, as some queries that may be inexpressible in a certain logic could merely be always true or false with the additional requirements put on the structure.

A logic able to express \( \text{PTIME} \) queries on programs would be useful in the field of compiler optimizations. Consider an optimization that takes exponential time to preform. However, say there was a logic that captured \( \text{PTIME} \) on programs, and there were a query in the logic that stated “This program can be further optimized”. Then given this query, an algorithm is guaranteed which runs in polynomial time and can test if the program can further be optimized by the aforementioned slow optimization. Thusly, the exponential time optimization need not be preformed if the program can not be further optimized.

3. RELATED WORK

Fagin’s theorem [2] demonstrates that existential second order sentences capture \( \text{NPTIME} \) queries. Fagin’s theorem shows that a certain logic, existential second order logic, expresses a query on graphs if and only if it is an \( \text{NP} \) query [1]. Because of this, if it were shown that \( \text{PTIME} \) did not have any such logic, this would show that \( \text{P} \neq \text{NP} \). Further, it also can give an idea of how to approach the problem of showing a logic can express some complexity class.

Further research has found new insights into \( \text{FOL} + \text{LFP} \). This extension still does not capture all \( \text{PTIME} \) queries, but it captures more than first order logic, as queries like connectivity are expressible. Furthermore, on certain structures it has been shown to capture \( \text{PTIME} \). Ordered graphs were the first structure on which \( \text{FOL} + \text{LFP} \) was found to express \( \text{PTIME} \) [5]. After that, other structures, such as planar graphs, trees, bounded treewidth graphs, have been shown to have all their \( \text{PTIME} \) queries expressible in \( \text{FOL} + \text{LFP} \). [4] These structures were proven to have this property, by showing that they have a unique ordering implicit in their structure, so they have been reduced to ordered structures, which encapsulate \( \text{PTIME} \) [7].

4. DEFINITIONS

Definition 4.1 (LexedJava). \textit{Java} is a programming language, it can be interpreted as a list of characters. \textit{LexedJava} is these characters lexed into tokens for each word in the program. Function and variable names are turned into \textit{fun} and \textit{var}. One could do this proof on Java as a list of characters. However, in the interest of additional clarity of the logic, the characters are lexed into tokens, and variable names are replaced by \textit{var}, in a way that 2 variables are both lexed into \textit{var} if and only if they are the same variable.

Definition 4.2 (P’). \( \mathcal{P}' \) is a class of structures with a relation \( R_\tau \) for each token of \textit{LexedJava}, \( \tau \), as well as \( S \) the successor relation. \textit{Var}, \textit{Fun}, and \textit{S} are binary relations, and all other relations are unary.

Definition 4.3 (PPL). \( \text{PPL} \) is \( \text{FOL} + \text{LFP} + C \) on the class of structures \( \mathcal{P}' \).

5. PROOF OVERVIEW

The proof will be given by a series of lemmas:

Lemma 5.1. There is a function that runs in polynomial time, \( \psi : \text{LexedJava} \rightarrow \mathcal{P}' \). Define \( \mathcal{P} \) as \( 1\text{m}(\text{LexedJava}) \). Further, there is a polynomial time \( \psi^{-1} : \mathcal{P} \rightarrow \text{LexedJava} \) such that \( \psi^{-1} \circ \psi = \text{id}_{\text{LexedJava}} \).

Lemma 5.2. There are polynomial time functions \( \lambda_1, \lambda_2 \) from Turing Machines to Turing Machines, where for a \( \text{PTIME} \) algorithm \( M \) on \textit{LexedJava}, \( M(p) = \lambda_1(M)(\psi(p)) \), and for a \( \text{PTIME} \) algorithm \( M \) on \( \psi(p) \), \( M(\psi(p)) = \lambda_2(M)(p) \).

Lemma 5.3. There is a formula \( \phi \in \text{PPL} \) such that \( \phi \) has 2 free variables, and \( \phi(x, y) \) defines a total order on all \( \mathcal{P} \). This demonstrates that \( \text{FOL} + \text{LFP} + C \) captures all queries on \( \mathcal{P} \).

Theorem 5.4. A sentence \( \phi \) is in \( \text{PPL} \) if and only if there is a Turing Machine \( M \) such that \( M(p) \) is true if and only if \( \psi(p) \models \phi \). Furthermore, given \( \phi \), the associated \( M \) can be calculated in polynomial time.

Proof. Let \( \phi \) be in \( \text{PPL} \). From Immerman–Vardi, there is a Turing Machine \( M \) such that \( M(p) \) is true if and only if \( p \models \phi \), for \( p \in \mathcal{P} \), and \( M \) is generated in polynomial time. Consider \( \lambda_1(M) \), which is generated in polynomial time, as \( \lambda_2 \) is a polynomial time function. \( \lambda_1(M)(p) = M(\psi(p)) \). So \( \lambda_2(M)(p) = M(\psi(p)) \) which is true if and only if \( \phi(x, y) \) defines a total order on all \( \mathcal{P} \). This demonstrates that \( \text{FOL} + \text{LFP} + C \) captures all queries on \( \mathcal{P} \). 

6. PROOF SPECIFICS

6.1. Lemma 5.1, Definition of \( \psi \psi^{-1} \)

Let \( p \) be a program in \textit{LexedJava}. We inductively define how to compute \( \psi \). If \( p = [] \) (empty list), then define the structure to be \( (\emptyset, \ldots, \emptyset) \). If \( p = t : p \) (list with head \( t \), rest of the list \( p \)), and \( \psi(p) = \langle x_1, \ldots, x_n \rangle, R_{t1}, \ldots, R_{var}, S \rangle \), then \( \psi(p') = \langle \{x_1, \ldots, x_{n+1}\}, \ldots, R_{t1}', \ldots, \overline{R}_{var}, S' \rangle \), where \( R_{t1}' \) is defined depending on what token \( x_{n+1} \) is. If \( x_{n+1} = t \), \( R_{t1}' = R_{t1} \). If \( x_{n+1} \) is a token, then for \( \tau \neq \text{tau}, \text{set } R_{\tau}' = R_{\tau} \cup \{x_{n+1}\} \). If \( \tau = \text{fun} \) or \( \text{var} \), then set \( R_{\tau}' = R_{\tau} \cup \{x_{n+1}, i\} \), otherwise set \( R_{\tau}' = R_{\tau} \cup \{x_{n+1}, i\} \). Intuitively, this makes each variable true of exactly 1 relation, the one corresponding to the token \( \tau \) is. \( S' = S \cup \{x_{n+1}, i\} \).

This is clearly a polynomial time algorithm, as it takes constant time for each token, so it takes linear time in the size of the lexed program. Furthermore, \( \psi^{-1} \) can be calculated in a similar way, given a structure of size \( n \), create a list of \( n \) elements, where the token at place \( i \) is \( \tau \) where \( R_{\tau} \) is the only relation that \( x_i \), the \( i \)th largest element is an element of. The fact these are inverses are clear by the definitions.
Further, every \textit{PTIME} algorithm that creates a \textit{PTIME} element, the element generating the \textit{PTIME} x element, merely expressing the query in this language, in the language. However, if the language were closer to natural language, merely expressing the query in this language would be sufficient to generate a \textit{PTIME} program. This is still true for \textit{PPL}, but expressing the query is harder in \textit{PPL} because it is so distinct from natural language.

Another problem with \textit{PPL} is its limit to queries. While being able to express all \textit{PTIME} queries is nice, expressing all \textit{PTIME} algorithms would be stronger. Were some extension of \textit{PPL} able to express all \textit{PTIME} algorithms, where a sentence in the language could express \textit{PTIME} algorithms, optimizations of problems, instead of merely the query about if optimizations are possible, would be able to be preformed.

Further, \textit{PPL} guarantees merely a \textit{PTIME} Turing Machine for any sentence. However, a simple query like “is 0 ever added to a variable” may take \(n^{100}\) or worse time. Further research could be put into transforming a sentence into the most efficient Turing Machine instead of merely a \textit{PTIME} Turing Machine.

Lastly, programs are important in many areas of computer science. Perhaps being able to express \textit{PTIME} queries on programs can provide an approach to \textit{PTIME} queries on arbitrary structures.

\section{ETHICS}

There is no foreseeable ethical dilemmas arising from \textit{PPL}. \textit{PPL} is merely a tool to assist in some proofs, and provide an alternate method of generating efficient queries on programs. There is no anticipated immoral uses for \textit{PPL}.

\section{CONCLUSIONS}

\textit{PPL}, defined as \textit{FOL} + \textit{LFP} + \textit{C} on \(\mathcal{P}\), can be shown to express all \textit{PTIME} queries on programs. Given a query \(\phi\) in \textit{PPL}, a Turing Machine \(M\) that runs in polynomial time can be generated in polynomial time where \(M(p)\) is true if and only if \(\psi(A) \models \phi\), where \(\psi\) is a pre-defined function from programs to structures. Conversely, given a Turing Machine \(M\) that runs in polynomial time, there is a query \(\phi\) in \textit{PPL} where \(\psi(A) \models \phi\) if and only if \(M(p)\). \textit{PPL} can be viewed as a programming language, except where most languages are Turing Complete (able to express all computable queries), the strength of this language is the limits of its expressibility. In most programming languages, the program’s existence does not demonstrate much in itself. However, if \textit{PPL} expresses a query, one knows that query has a \textit{PTIME} algorithm.

Thusly, if one wants to demonstrate a query can be efficiently solved, it suffices to express that query in \textit{PPL}. In this manner, \textit{PPL} can be used as a proof technique. Further, given a question about programs, if one expresses this in \textit{PPL}, by a set \textit{PTIME} algorithm, a program can be generated that answers the question efficiently. Further, if there is a \textit{PTIME} query on programs, then there will never be a situation where \textit{PPL} will not have a \(\phi\) that generates that \textit{PTIME} query.

\section{REFERENCES}


http://www.informatik.uni-leipzig.de/~lohrey/.